Woche 11 – Übersicht & Tricks

(Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.)

Anmerkungen zu Serie 9 - Allgemeines:

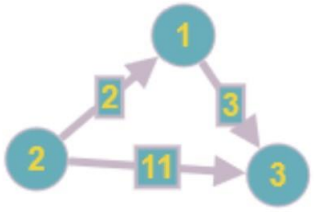
- Die Aufgaben wurden insgesamt sehr gut gelöst (Bis jetzt beste Woche)
- Anmerkung zu Pseudocode: Wenn ihr Ergebnisse von rekursiven Aufrufen verwenden wollt, müsst ihr das auch im Pseudocode deutlich machen. Einfach den rekursiven Funktions-Aufruf hinzuschreiben, reicht nicht. Anstatt dessen eventuell (je nach Kontext) ein «return» davor schreiben oder den rekursiven Aufruf in einer expression verwenden.

Shortest Paths Probleme – Einführung:

Wir erweitern unsere Definition von Graphen von $G = (V, E)$ zu $G = (V, E, c)$ wobei $c: E \to \mathbb{R}$ die sogenannte “Kantengewichtsfunktion” ist, die jeder Kante eine reelle Zahl zuordnet. Diese Zuordnung beschreibt üblicherweise irgendwelche Kosten oder Längen, um Unterschiede zwischen Kanten zu modellieren, beispielsweise könnte die Kantengewichtsfunktion die Distanz in Kilometern zwischen zwei, per Autobahn verbundenen, Städten definieren, wenn Knoten Städte und Kanten Autobahnen wären. Formal schreiben wir: $c\big((u,v)\big) = Gewicht\ der\ Kante\ (u,v) \in E$.

Des Weiteren definieren wir die **Distanz** zwischen zwei beliebigen Knoten als: $d\big((s,t)\big) = Laenge\ eines\ kürzesten\ Pfades\ zwischen\ s\ und\ t$ (wobei hier als Länge die Summe der Kantengewichte gemeint ist).

Beispiel:



Hier ist die Distanz von 2 zu 3 der Wert 5, obwohl der Pfad dieser Länge mehr Kanten verwendet als der Pfad der nur eine Kante, direkt von 2 zu 3 nimmt.

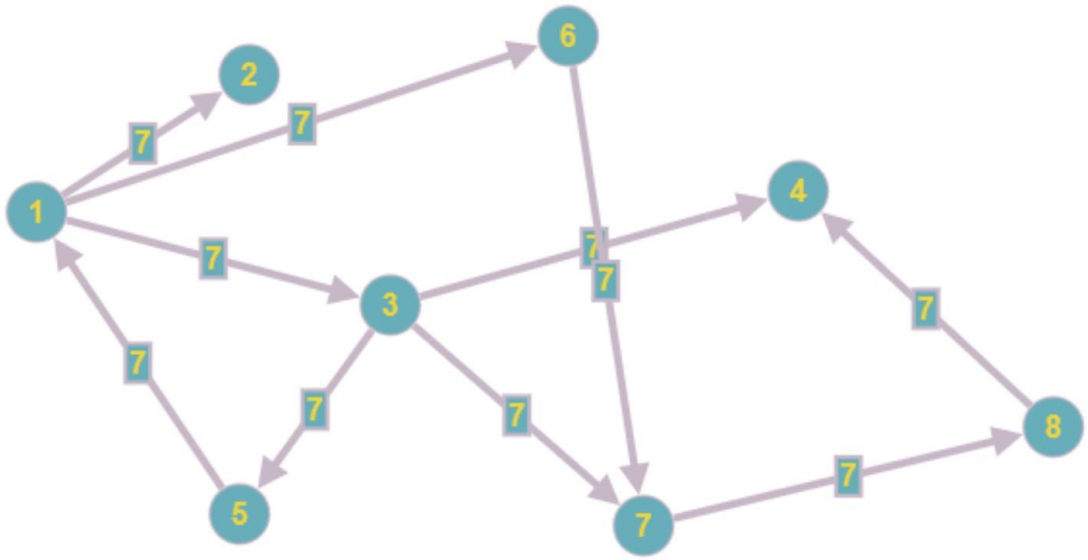
Wir sprechen ausserdem häufig vom “Single-Source” oder “One-to-All” Shortest-Paths-Problem (SSSP). Dabei ist gemeint, dass wir für einen gegeben Knoten die Distanzen zu allen anderen Knoten des Graphen bestimmen wollen.

Shortest Paths Probleme – Bekanntes:

In den letzten beiden Wochen haben wir das Shortest-Path-Problem schon in bestimmter Form gesehen: Für “Directed Acyclic Graphs” (DAGs), also Graphen, die eine topologische Sortierung besitzen, können wir das shortest Paths Problem mit DP (dist[i] = min{dist[j] + c((i,j)), dist[i]}) berechnen (in $O(|E| + |V|)$, da wir jede Kante und jeden Knoten exakt einmal betrachten).

Ausserdem haben wir schon gesehen, dass wir den BFS-Algorithmus verwenden können, um kürzeste Pfade auf beliebigen Graphen **ohne Kantengewichte** zu bestimmen. Dieser läuft auch in $O(|V| + |E|)$ (siehe letzte Woche) und ist damit für diesen Fall der effizienteste Algorithmus, der in dieser Vorlesung vorgestellt wird.

Trick-Question: Wie können wir unser bisherigen Wissen verwenden, um das single-source-shortest-paths Problem für diesen Graphen zu lösen?



Ganz einfach: Wenn wir Wissen, wie wir das SSSP-Problem für Graphen ohne Kantengewichte lösen können, können wir die gleiche Idee auch für Graphen mit uniformen Kantengewichten verwenden. Dafür teilen wir einfach das Kantengewicht jeder Kante durch den Gleichen Wert (sodass jede Kante anschliessend Kantengewicht 1 – sprich eine gewöhnliche Kante ist – hat). Sollen wir die ermittelten Distanzen auch angeben (und nicht nur die Pfade bestimmen), müssen wir uns am Ende nur daran erinnern, die bestimmte Distanz mit 7 zu multiplizieren.

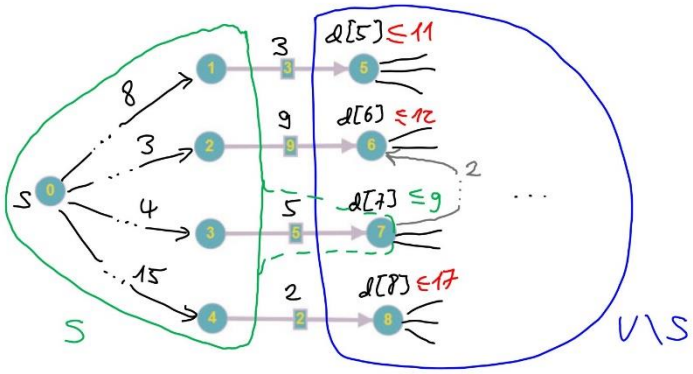
Dijkstra-Algorithmus (NICHT-NEGATIVE KANTENGEWICHTE):

Vorraussetzung: Ein Graph mit beliebigen, aber nicht-negativen Kantengewichten. Der Dijkstra-Algorithmus beruht auf folgender Idee: Wir vergrössern die Menge $\boldsymbol{S}$ der Knoten, für die wir die kürzeste Distanz von s aus kennen, induktiv, bis diese Menge alle Knoten umfasst.

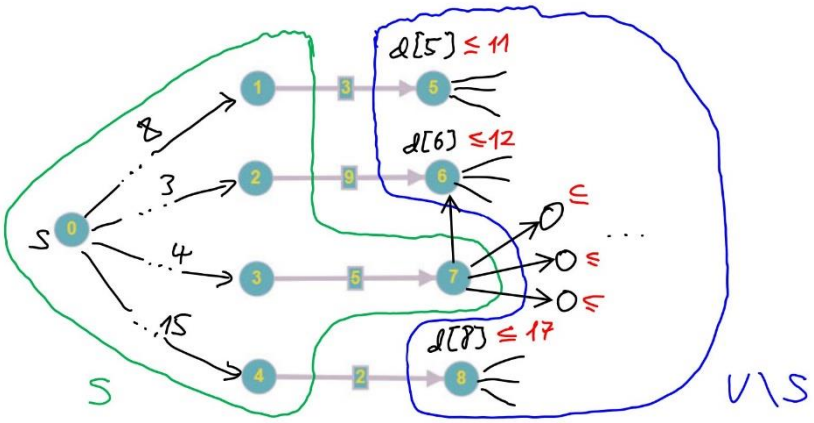
Dafür merken wir uns zunächst die Distanzen von s zu allen anderen Knoten in einem "Distance"-Array, sprich $\boldsymbol{d}[v]$ speichert die Distanz von s zu v. Anfangs setzen wir diese Distanz für alle Knoten auf $d[v] = \infty$ und für $d[s] = 0$. Am Ende soll in $d[v]$ die Distanz von s zu v für alle $v \in V$ stehen. Ausserdem speichern wir uns für jeden Knoten in einem "Parent"-Array seinen Vorgänger in einem kürzesten Pfad, d.h. $\boldsymbol{p}[v]$ speichert den Vorgänger in einem kürzesten Pfad von s zu v.

Simpel gesagt funktioniert der Algorithmus jetzt folgendermassen: In jedem Schritt suchen wir uns den Knoten v mit der geringsten Distanz von s (minimum $d(s, u)$) an aus, der noch nicht in $\boldsymbol{S}$ ist. Dann fügen wir ihn zu $\boldsymbol{S}$ hinzu und aktualisieren für alle, von ihm erreichbaren Nachbarn $u \in N(v)$ die Distanzen mit $d[u] = \min\{d[u], d[v] + c((v, u))\}$. Falls es für einen Knoten u einen kürzeren Pfad über v gibt, aktualisieren wir: $p[u] = v$. Dann wiederholen wir den Schritt für den jetzt billigsten Knoten, der nicht in $\boldsymbol{S}$ ist (v ist ja nun in S). Nun zeige ich kurz, warum das funktioniert:

Nach i Iterationen kennen wir die kürzesten Pfade (und damit auch die finalen Distanzen) zu allen Knoten im grünen Bereich. Es kann keine kürzeren Wege zu diesen Knoten von s aus geben, da alle Kanten nicht-negative Gewichte haben und jeder Pfad, der von grün zu blau und dann wieder zu grün zurückführt, mindestens die Kosten 0 hat.



Aber wir wissen auch: Um unsere grüne Menge S (Knoten, zu denen wir die kürzesten Pfade schon kennen) zu vergrössern, müssen wir eine der Kanten, die S und den Rest der Knoten, also $V\backslash S$, verbinden, verwenden. Dafür nehmen wir die, die zur billigsten Distanz führt: In diesem Fall die Kante $(3,7)$ (mit Kosten 5). Denn jeder andere Pfad von s nach Knoten 7 ist mindestens so lang bzw. länger als der mit Kante $(3,7)$, also können wir den Knoten 7 mit berechneter Distanz $d((0,7)) = 9$ zur Menge S hinzufügen.



Nach exakt diesem Prinzip verfahren wir nun weiter, bis wir für alle Knoten die Distanz von s aus bestimmt haben.

Um effizient zu verwalten, welche Knoten in S liegen, welche in $V\backslash S$ und welche davon die billigsten sind, verwenden wir und eine, uns bereits bekannte Datenstruktur: Die Priority-Queue, allerdings mit Min statt Max, verwaltet alle Knoten in $V\backslash S$. Das hinzufügen eines Knoten v zu S entspricht also dem Entfernen eines Knoten aus der Priority-Queue.

Insgesamt gehen wir also folgendermassen vor:

Zunächst legen wir eine leere Priority-Queue Q an und fügen unseren Startknoten s in die Queue ein. Dann arbeiten wir in einer while-schleife, die so lange läuft, bis die Queue leer ist:

Wir extrahieren das minimale Element von Q und betrachten alle seine Nachbarn $u \in N(v)$: Falls $p[u]$ noch null ist, wurde u noch nie vorher entdeckt. In diesem Fall setzen wir $d[u] = d[v] + c((v,u))$ und $p[u] = v$ und fügen u in Q ein, das heisst, u ist in $V\backslash S$, da es noch kürzere Pfade zu u

geben könnte. Falls $p[u]$ nicht mehr null ist, prüfen wir, ob $d[v] + c\big((v,u)\big) \leq d[u]$ (und aktualisieren es, falls es dem so ist, inklusive des parent-eintrags von u auf $p[u] = v$). Dann müssen wir natürlich aber auch den distanz-eintrag von u in Q auf $d[v] + c((u,v))$ aktualisieren.

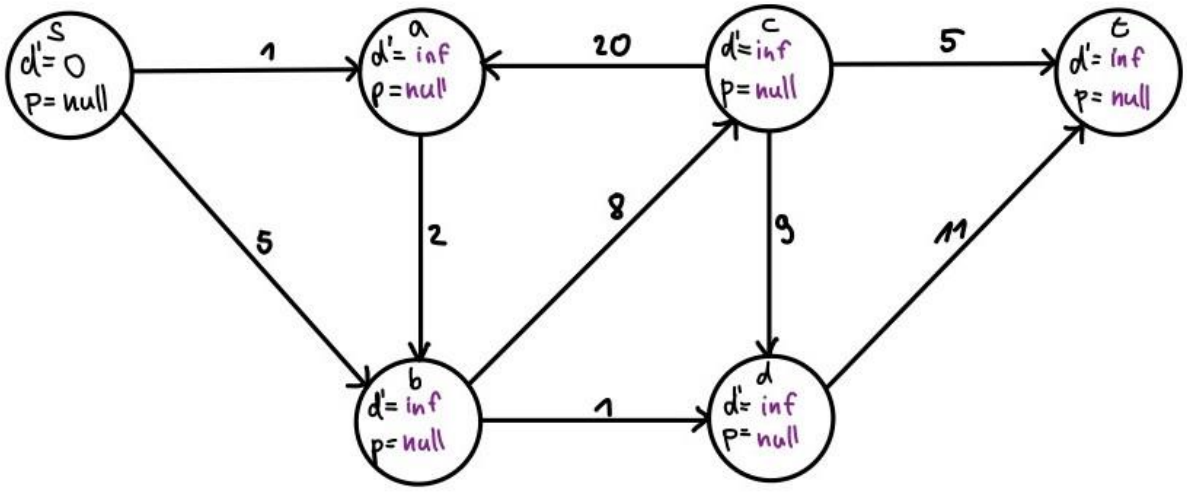
Im Skript sieht das ganze (in Pseudocode) so aus:

```
DIJKSTRA(G = (V, E), s)
 1 for each v ∈ V\{s} do                    ▷ Initialisiere für alle Knoten die
 2     d[v] ← ∞; p[v] ← null                ▷ Distanz zu s sowie Vorgänger
 3 d[s] ← 0; p[s] ← null                    ▷ Initialisierung des Startknotens
 4 Q ← ∅                                    ▷ Leere Prioritätswarteschlange Q
 5 INSERT(s, 0, Q)                          ▷ Füge s zu Q hinzu
 6 while Q ≠ ∅ do
 7     u ← EXTRACT-MIN(Q)                   ▷ Aktueller Knoten
 8     for each (u, v) ∈ E do               ▷ Inspiziere Nachfolger
 9         if p[v] = null then              ▷ v wurde noch nicht entdeckt
10             d[v] ← d[u] + w((u, v))      ▷ Berechne obere Schranke
11             p[v] ← u                     ▷ Speichere u als Vorgänger von v
12             ENQUEUE(v, d[v], Q)          ▷ Füge v zu Q hinzu
13         else if d[u] + w((u, v)) < d[v] then ▷ Kürzerer Weg zu v entdeckt
14             d[v] ← d[u] + w((u, v))      ▷ Aktualisiere obere Schranke
15             p[v] ← u                     ▷ Speichere u als Vorgänger von v
16             DECREASE-KEY(v, d[v], Q)     ▷ Setze Priorität von v herab
```

Das mag alles sehr verwirrend wirken, daher hier ein ausführliches Beispiel:

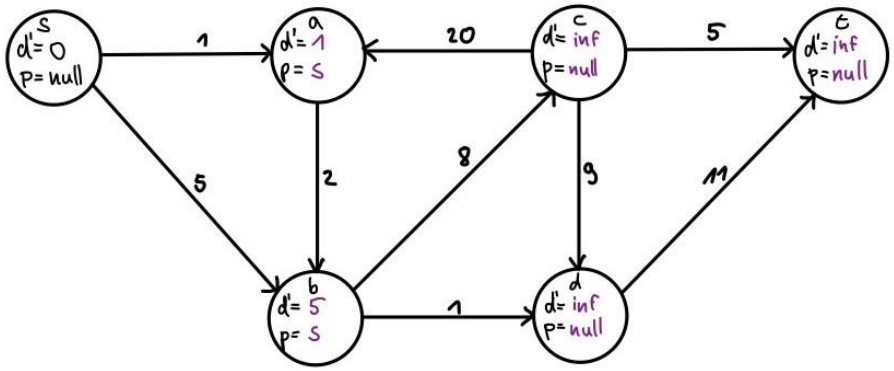
Wir wollen für folgenden Graphen die kürzesten Distanzen von s zu allen anderen Knoten berechnen. Dafür habe ich die Eintrage der Arrays (damit es leichter zu sehen ist) direkt in die Knoten eingeschrieben. Wir oben erklärt, setzen wir anfangs alle parent-einträge auf null und alle distanzeinträge auf unendlich (bis auf den von s, den setzen wir auf 0).

Wir fügen s in Q ein, d.h.: $Q = \{(s, 0)\}$



Jetzt sind wir im While-Loop und entfernen das minimal Element aus der Priority-Queue, also in diesem Schritt s. Wir betrachten alle Nachbarn, setzen die Distanzen und Parent-Einträge und fügen

sie in Q ein. Ausserdem können wir s in S einfügen:

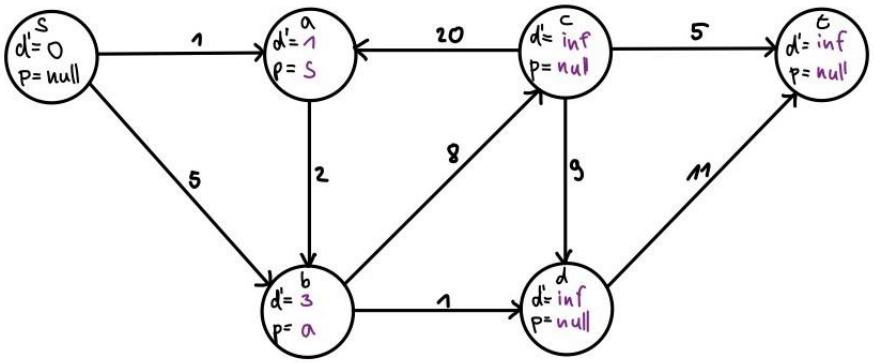


$S = \{s\}$

Jetzt sieht Q so aus: $Q = \{(a, 1), (b, 5)\}$

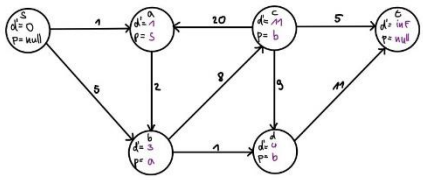
Also extrahieren wir das minimale Element, a und betrachten alle Nachbarn. Das ist nur b. Da $d[a] + c\big((a, b)\big) = 3 \leq 5 = d[b]$, aktualisieren wir die Einträge von b zu $p[b] = a$ und $d[b] = 3$.

Ausserdem können wir jetzt auch a zu S hinzufügen, weil es von s aus keinen kürzeren Pfad zu a geben kann.

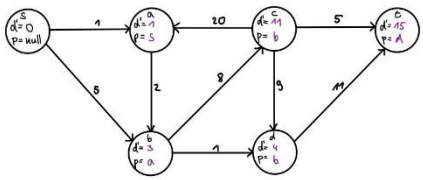


$S = \{s, a\}$

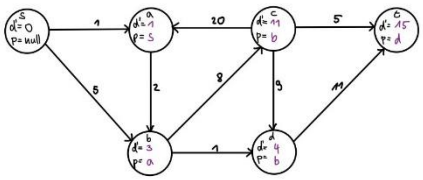
Nach dem gleichen Muster verfahren wir nun, bis alle Distanzen bestimmt sind:



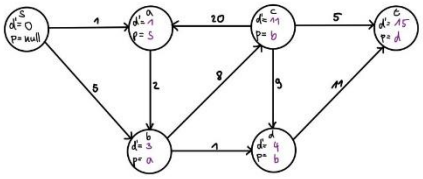
$S = \{s, a, b\}$



$S = \{s, a, b, d\}$



$S = \{s, a, b, d, c\}$



$S = \{s, a, b, d, c, t\}$

Insgesamt läuft der Dijkstra-Algorithmus in $O((|V| + |E|) * \log |V|)$. Man kann in mit sogenannten Fibonacci-Heaps noch weiter verbessern (siehe Skript).

Bellman-Ford Algorithmus:

Da Dijkstra nur für Graphen mit nicht-negativen Kantengewichten funktioniert, benötigen wir einen weiteren Algorithmus, der dieses Problem löst: Dies ist der Bellman-Ford Algorithmus. Falls es negative Kreise gibt, ist dieser sogar in der Lage, diese zu erkennen (interessant für Arbitrage-Handel).

Der Algorithmus funktioniert folgendermassen: Wir betrachten $|V| - 1$ mal (denn ein Pfad kann höchstens aus $|V| - 1$ Kanten bestehen) alle Kanten und versuchen jedes Mal, die je betrachtete Kante (u, v) zu verwenden, um die Distanz von s zu v zu verringern.

Dazu initialisieren wir zunächst ein Distanz-Array $d[]$ mit $d[v] = \infty$ für alle Knoten ausser s, da setzen wir wieder $d[s] = 0$. (Falls notwendig, weil die konkreten Pfade gesucht sind, können wir auch noch ein $parent$-array verwenden).

Nun gehen wir in jeder der $|V| - 1$ Iterationen über alle Kanten $(u, v) \in E$ und setzen folgendes:

$$d[v] = \min\{d[v], d[u] + c((u, v))\}$$

Als Algorithmus sieht das im Skript (Pseudocode) so aus:

BELLMAN-FORD$(G = (V, E), s)$

```
1  for each v ∈ V\{s} do                    ▷ Initialisiere für alle Knoten die
2      d[v] ← ∞; p[v] ← null                ▷ Distanz zu s sowie Vorgänger
3  d[s] ← 0; p[s] ← null                    ▷ Initialisierung des Startknotens
4  for i ← 1, 2, ..., |V| − 1 do            ▷ Wiederhole |V| − 1 Mal
5      for each (u, v) ∈ E do               ▷ Iteriere über alle Kanten (u, v)
6          if d[v] > d[u] + w((u, v)) then  ▷ Relaxiere Kante (u, v)
7              d[v] ← d[u] + w((u, v))      ▷ Berechne obere Schranke
8              p[v] ← u                     ▷ Speichere u als Vorgänger von v
9  for each (u, v) ∈ E do                   ▷ Prüfe, ob eine weitere Kante
10     if d[u] + w((u, v)) < d[v] then      ▷ relaxiert werden kann
11         Melde Kreis mit negativem Gewicht
```

Wir sehen: Am ende iterieren wir noch einmal über die Kanten und gucken, ob es möglich ist, immernoch einen kürzeren Pfad zu finden. Wir wissen, dass ein Pfad in einem Graphen aus $|V|$ Knoten höchstens aus $|V| - 1$ Kanten bestehen kann. Ist also dennoch möglich, einen kürzeren "Pfad" zu finden, muss es einen negativen Kreis geben.

Der Algorithmus läuft ziemlich offensichtlich in $O(|V| * |E|)$

Hier ist ein schönes Beispiel:

https://www.youtube.com/watch?v=Pv9tQ7FatGY